Davies, Steven and Roper, Marc (2014) What's in a bug report? In: IEEE International Symposium on Empirical Software Engineering and Measurement. ACM. ISBN 9781450327749 , http://dx.doi.org/10.1145/2652524.2652541

What's in a Bug Report?

Steven Davies
Dept. Computer and Information Sciences
University of Strathclyde
Glasgow, U.K.
s.davies@strath.ac.uk

Marc Roper*
Dept. Computer and Information Sciences
University of Strathclyde
Glasgow, U.K.
marc.roper@strath.ac.uk

ABSTRACT

Context: Bug reports are the primary means by which users of a system are able to communicate a problem to the developers, and their contents are important - not only to support developers in maintaining the system, but also as the basis of automated tools to assist in the challenging tasks of finding and fixing bugs.
Goal: This paper aims to investigate how users report bugs in systems: what information is provided, how frequently, and the consequences of this.
Method: The study examined the quality and quantity of information provided in 1600 bugs reports drawn from four open-source projects (Eclipse, Firefox, Apache HTTP, and Facebook API), recorded what information users actually provide, how and when users provide the information, and how this affects the outcome of the bug.
Results: Of the recorded sources of information, only *observed behaviour* and *expected results* appeared in more than 50% of reports. Those sources deemed highly useful by developers and tools such as *stack traces* and *test cases* appeared very infrequently. However, no strong relationship was observed between the provided information and the outcome of the bug.
Conclusions: The paper demonstrates a clear mismatch between the information that developers would wish to appear in a bug report, and the information that actually appears. Furthermore, the quality of bug reports has an important impact on research which might rely on extracting this information automatically.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.2.6 [**Software Engineering**]: Programming Environments—*integrated environments*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*documentation*

*Corresponding author.

ESEM'14 September 18–19, 2014, Torino, Italy

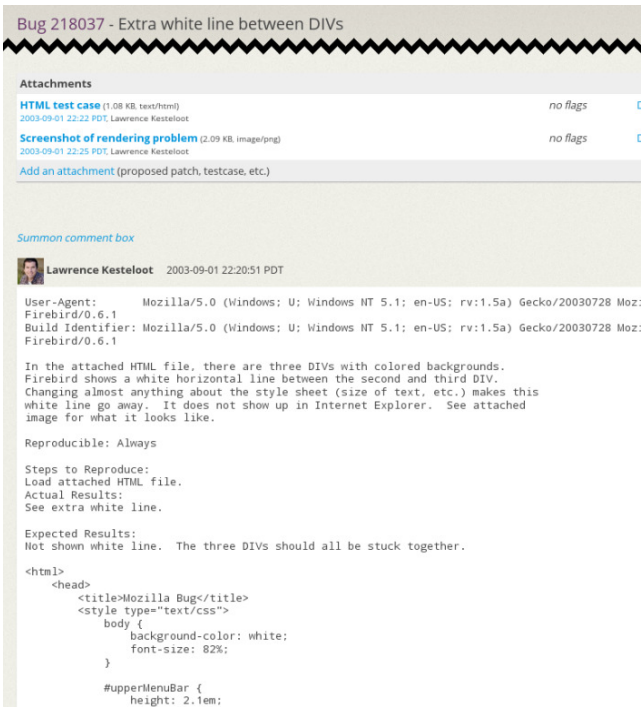

Figure 1: Mozilla Bug 218037 (Cropped)

1. INTRODUCTION

Figure 1 shows an extract from Mozilla [22] Bug 218037, and is in many ways the perfect example of how a bug should be filed. The report is detailed, and contains clear information about what happened and what the user expected to happen. It goes on to contain an extensive example to assist developers in reproducing the bug, and information about the user's environment when they encountered the bug. As long as this information is accurate, this sort of bug report should be reasonably straightforward for a developer to fix.

Unfortunately, not all bug reports are quite so well-written as Mozilla Bug 218037. A stark contrast can be seen in the bug report for Apache [1] Bug 25091 shown in Fig. 2. This bug report is very minimal, containing just the terse statement, "`break file download of large file (>4 mbyte)`". Indeed, it is not clear whether this is a bug or a user error. The developer makes a suggestion which may solve the problem and politely asks for more information (also indicating how the users could obtain this). A few days later, in the

Figure 2: Apache Bug 25091

light of no response, they prompt the user again for this information, but to no avail. Finally the developer chooses to close the bug as invalid - all effort associated with this bug has been wasted.

The way in which a bug is reported is clearly of importance to the developer charged with fixing the bug, as it can have a big impact on the ease with which they may be able to fix the bug. However, this information is also of wider interest and impacts on a number of important activities from mining bug repositories for information, through developing tools to support activities such as bug localisation, to building the next generation of bug tracking systems. Knowledge of *what* information that is reported, the *way* in which it is reported and *how frequently* it appears will have a significant influence on all these activities. Unfortunately, this information - particularly the last point - seems very hard to come by.

This paper reports on an empirical investigation of 1600 bugs in 4 open-source projects, examining the quantity and quality of information provided by users when they are reporting bugs. In particular, it looks at what information developers want in bug reports (as identified by Bettenburg et al. [3]) compared to what information users actually provide, how and when users provide the information, how this affects the outcome of the bug, and how this information can be used to improve the bug fixing process. The main findings from this investigation are that in many cases, bug reports are neither complete nor accurate, and often do not provide all the information that developers find useful when fixing bugs. Furthermore, it reveals that there would be a number of difficulties in automatically extracting relevant information from bug reports.

2. WHAT INFORMATION IS USEFUL TO DEVELOPERS WHEN FIXING BUGS?

One of the overall aims of examining the information provided in bug reports is to identify ways in which the bug fixing process can be improved. One potential first step in doing so is to examine what information developers themselves want when fixing bugs. Bettenburg et al. [3] surveyed 156 developers of Apache, Eclipse [13] and Mozilla about what sections of a bug report they found most useful when fixing bugs. In order, the ten most important features (from a total of 16 options) were reported as:

1. ***Steps to reproduce:*** A clear set of instructions that the developer can use to reproduce the bug on their own machine
2. ***Stack traces:*** A stack trace produced by the application, most often when the bug is reporting a crash in the application
3. ***Test cases:*** One or more test cases that the developer can use to determine when they have fixed the bug
4. ***Observed behaviour:*** What the user saw happen in the application as a result of the bug
5. ***Screenshots:*** A screenshot of the application while the bug is occurring
6. ***Expected behaviour:*** What the user expected to happen, usually contrasted with *Observed behaviour*
7. ***Code examples:*** An example of some code which can cause the bug
8. ***Summary:*** A short (usually one-sentence) summary of the bug
9. ***Version:*** What version of the application the user was using at the time of the error
10. ***Error reports:*** An error report produced by the application as the bug occurred

Similar results were found in a survey of Microsoft developers which was carried out to determine which features most influenced whether or not a bug would eventually be fixed [17].

Unfortunately, the information developers want is not always provided by users, and various research has reported that *Screenshots* and *Stack traces* [4], patches (which are proposed solutions to the bug) [18] and *Code examples* [3] have all been found to be relatively uncommon in particular projects. This behaviour does not go unnoticed by developers; one survey suggested only around half feel that bug reports are nearly always complete [20]. Additionally, Breu et al. [6] looked at the questions asked during the resolution of bug reports, and found a significant proportion of these were related to missing or inaccurate information. An in-depth study of bugs at Microsoft, along with a more general survey of employees [2], found that the majority of bug reports were missing information, and a significant proportion contained inaccurate information. Ko and Chilana [19] found found similar results – the majority of bug reports

by normal users of the Mozilla bug tracking system (BTS) contributed no useful information.

Some studies have investigated which attributes can be used to help predict the fix time of the bug [18], or the developers who should be responsible for fixing it [9]. There have also been attempts at determining how relevant *Stack traces* are to fixing bugs [23], and to whether *Error reports* can be used to identify the causes of bugs [28]. However, no study has attempted to show how commonly these various features occur in bug reports.

3. INVESTIGATION

To investigate how often the features desired by developers were actually provided in bug reports, 1600 bugs across 4 projects were examined to determine how often users provide each part of a bug report. The study also examined various issues surrounding the relationships between different aspects of a bug report, the variance between projects and the potential for automatic extraction of data from bug reports.

3.1 Subjects

The four projects involved in the investigation were:

Eclipse: An open-source integrated development environment (IDE) and application framework, written in Java

Firefox [15]**:** An open-source browser, written mainly in C++

Apache HTTP: An open-source HTTP server, written in C

Facebook [14] **Application Programming Interface (API):** Proprietary APIs for a social network, available in several languages[1]

A number of scripts were developed to extract a random subset of 400 bugs from each of the repositories. These projects represent a variety of different uses and languages and attract user populations of differing size and technical experience. Whilst three of the projects are open-source and one is closed-source, each of the projects in question makes use of an open bug repository. Anyone can report and comment on bugs about the project, and take part in conversations about the bugs with developers.

3.2 Features

While the exact implementation varies between systems, bug reports usually consist of a number of fields. Some of these fields, such as the severity or version, can only take a limited number of values. Information can be extracted from these fields in a relatively straightforward manner using automated techniques. However, there are a number of other pieces of information which are desirable in a bug report that are not contained within these fields, such as the *Expected behaviour* or *Steps to reproduce*. Unfortunately, in general BTSs have no specific support for any of these features, and they are usually provided (if indeed they are provided at all) as part of the title, description or comments, all of which are unstructured plain text, or as generic attachments. As such, identifying how often they are provided is not straightforward; there are no simple techniques for automatically extracting them.

For each bug in the sample, the basic structured information available was recorded. The unstructured information for each bug report was then manually examined by the first author to identify whether any of the following features, based on those listed in Section 2, were present in the bug report, and how they were reported. While amongst some projects it was common to explicitly label some features, e.g. "Expected results: ...", the investigation did not rely on these. A description of what the reporter expected to happen was sufficient, and similar procedures were followed for other features. Since this obviously leads to the evaluation being somewhat subjective, a number of judgement calls had to be made. The most common and important decisions are detailed alongside the relevant feature below, but this is not a complete list.

The features examined (based on the top ten identified by [3]) were:

***Observed behaviour* (Obs):** In general, a statement which simply says a particular feature 'does not work' or something similar was not sufficient to consider *Observed behaviour* as present. Users had to provide at least a minimal amount of detail about what happened or what they saw.

***Expected behaviour* (Exp):** Similarly, while a statement such as 'I received the following error' is sufficient for *Observed behaviour*, it was not sufficient for *Expected behaviour*. The user had to make clear that they expected not to receive any error (or indeed, that they were expecting an error but that the error they received was incorrect). However, feature requests or patch descriptions were often made in an imperative form and this was considered as sufficient for *Expected behaviour*.

***Steps to reproduce* (Rep):** Enough detail had to be provided to allow the bug to be reproduced on another machine, although this did not have to be explicitly given as a numbered sequence of instructions.

***Error reports* (Err):** *Error reports* included stack traces, and any time when a quoted error message was provided, as well as more detailed logs or Java core dumps. Apache logs with error or critical notices were considered *Error reports* as were other levels of logs if they clearly contained an error message. *Error reports* were counted even if the text appeared only in a screenshot.

***Stack traces* (Sta):** *Stack traces* did not have to be from an exception. For example, backtraces obtained through gdb [16] were sufficient. The output from strace [25], truss [26] and similar tools were not considered to be *Stack traces*, as these only contained the system calls being made, not the application code itself. In many ways, *Stack traces* is a more specific form of *Error reports*, which is itself a more specific form of *Observed behaviour*.

***Screenshots* (Scr):** Only *Screenshots* (or videos) of the application in question were considered, not of other applications or of proposed changes.

[1]Note that this is intended for developers of Facebook applications, not everyday users of the site

***Code examples* (Cod):** *Code examples* were not required to be either complete or minimal. In many cases, users provided links to webpages or Facebook applications which reproduced the bug in questions. These were considered *Code examples*. Apache configuration directives and files were considered *Code examples*. Many features of Facebook could be exercised by simple one-line RESTful URLs. These were also considered *Code examples*.

***Test cases* (Tes):** *Test cases* had to be self-contained and automatic. No manual action should be needed after setup, but some interpretation of results was considered acceptable e.g. a test that prints 'Success' or 'Failure'.

***Build information* (Bui):** This field records not just the release version of code being used, but also records if the user was using a 'between-release' version of the code, as well as what options were set when they built the software. *Build information* is specified differently for each project.

***Application Code* (App):** This was used to record any time when particular lines or methods of the project were identified as being the source or solution of a problem, including patches. Note that this is code from the source of the applicable project, as opposed to *Code examples* which is code written by users that is a client of the project code.

The assessment did not take into consideration whether the information for given features was accurate, nor determine its quality. For example, if a bug report contained instructions to reproduce an issue, then this was marked as containing *Steps to reproduce*, even if a developer pointed out that these steps were not sufficient or correct.

4. WHAT INFORMATION DO USERS PROVIDE?

Figure 3 shows the number of bugs that contain each feature at some point in the bug report. As shown, *Observed behaviour* is found in the vast majority of reports, with *Expected behaviour* in more than half and *Steps to reproduce* in slightly less. Barring some highly unusual bugs, it *should* be possible to provide all these fields in any bug report, so the fact that they are not provided more often could be seen as disappointing. This is especially true for *Steps to reproduce*, as developers consider this the single most important feature needed when handling a bug report.

The other features are found in far fewer reports, with *Screenshots*, *Stack traces* and *Test cases* all being found in less than 10% of bug reports. This is in line with results found by other researchers [3, 4]. These features are not necessarily appropriate for all bug reports however. For example, we do not know how many bugs without *Stack traces* did not generate a stack trace, and how many did generate a stack trace but the user chose not to report it. The low numbers of these features suggest however that bug fixing tools or techniques that rely on the presence of the features are not likely to be widely applicable.

4.1 Differences between projects

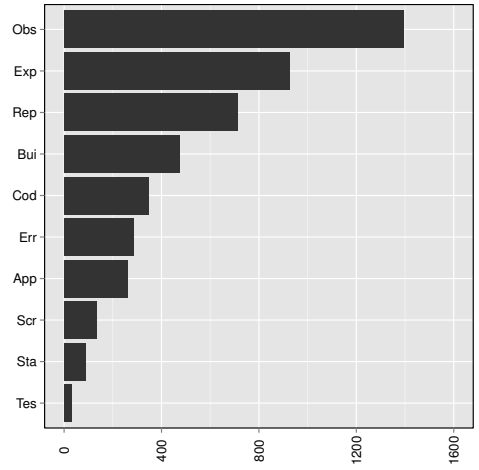

Figure 3: Number of bugs in which each feature was present

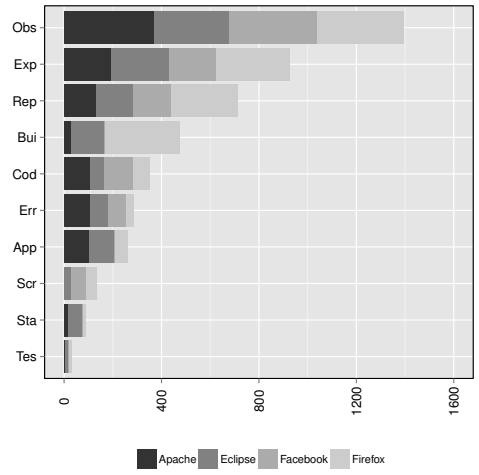

Figure 4: Number of bugs in which each feature was present, by project

As with many other measures in software engineering, it seems likely that the information provided in bug reports will vary between projects. Figure 4 shows the number of bug reports with each feature by project. It is clear from this that for some features, they are more common for some projects than for others. χ^2-tests for independence were carried out to test whether the presence or non-presence of each feature is dependent on the project. For each feature, p-values for the χ^2-tests were all less than 0.01, suggesting that there is indeed a relationship between the project and the presence of each feature.

Firefox is responsible for a significant proportion of the bugs with *Build information*, *Expected behaviour* and *Steps to reproduce*. One possible reason for that may be that Firefox has an optional interface that specifically prompts its users for this information (along with *Observed behaviour*) in a separate field for each, while the other projects simply expect the reporter to include them in the description (al-

though they may have documentation which tells reporters that all of them are expected). This suggests that such an interface could boost the numbers of bug reports which contain these fields if a similar interface was adopted by other projects.

Unsurprisingly given the nature of the application, Apache bugs contained virtually no *Screenshots* and very few *Test cases*, *Stack traces* or *Build information*. *Error reports*, and in particular error logs, are common however.

As no build identifiers are available to users, Facebook bugs contain no *Build information*. Also, likely due to the fact that the source code is not available to users, very few bug reports contain information about where in the code the bug is likely to be, and automated *Test cases* and *Stack traces* are also rare. Perhaps as a consequence of this, *Screenshots* and *Code examples*, often in the form of links to Facebook applications, are more common.

There are perhaps surprisingly few *Code examples* provided in Eclipse, given that it is after all an IDE. However *Stack traces* are more common than in other projects. This is likely due to the fact that they are more prominent within the application. Most errors within Eclipse will produce stack traces either in a dialog to users, in the error log, or in the Java core dump that may be produced by a crash. This is in contrast to Apache or Firefox, where the user must usually go to some manual effort, for example using gdb, to produce a stack trace for an error.

4.2 Amount of features

As well as the individual features, it is interesting to look at the overall number of features provided in each bug report, and how this differs by project. Figure 5 shows the number of features (out of the possible ten) contained in each bug report. As shown here, very few bugs have more than five features, and the majority have three or fewer. It appears that Facebook users appear to contribute less information on average, while Firefox users contribute more. The reasons for this are unclear, although it may be that bug reports on Facebook are less technically savvy, and a small minority of the bugs reported on Facebook were not bugs with the platform, but were instead with the website, and by users who are not likely to be used to BTSs.

As also shown in Fig. 5, a number of bug reports actually contained no useful information at all. Often, this was because the BTS was being entirely misused by users. Examples included: Firefox Bug 333204, Facebook Bugs 11460 and 13181, and Eclipse Bugs 157392 and 229028, all of which contained no or meaningless content; Firefox Bug 577566 and Facebook Bug 9843 which were in a foreign language, and so are less likely to be understood by the developers; Firefox Bug 643736 which was actually a bug in a specific website, and which had 'Chrome/IE' (i.e. two other browsers) listed as the build identifier; Facebook Bug 5718 which contained a job application; and Facebook Bug 3551 which simply contained a picture of some flowers. These types of bugs are the minority however, which should be somewhat encouraging for application developers. However, handling these bugs still incurs some cost for developers. Whilst some are relatively harmless and can be closed swiftly, others may require the developer to engage in a reasonable level of investigation before concluding that the bug is not valid.

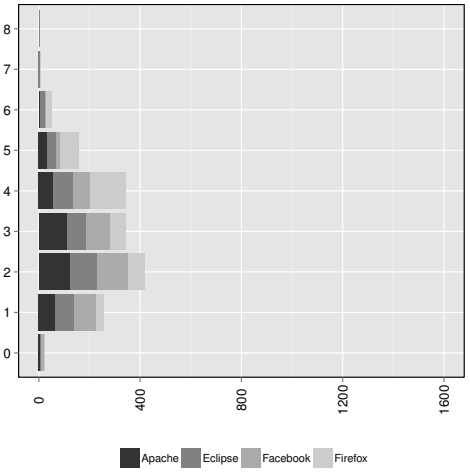

Figure 5: Total number of features contained in each bug report

4.3 Relationship between features

As well as examining the individual features, it is interesting to look at how many are provided together with one another. Table 1 shows for each feature the correlation between it and each other feature. Not all of the correlations shown are significant ($p<0.01$); those that are significant are highlighted in bold. While many of the correlations are indeed significant, none are particularly strong. There is a medium level of correlation between *Stack traces* and *Error reports*, as would be expected as one often contains the other. The same is true for *Code examples* and *Steps to reproduce*, as code samples are often the means of providing the necessary steps. Of some interest too is the slight negative correlation between *Observed behaviour* and *Expected behaviour*. This is due to almost all of the bugs which do not provide *Observed behaviour* providing *Expected behaviour*. Indeed, only 35 bugs did not contain at least one of the two features, which is encouraging as a bug report without either would be nearly impossible for a developer to resolve.

5. HOW DO USERS PROVIDE INFORMATION?

Figure 6 shows how the information required is provided: in the main body of the report, as an attachment to the report, or in an external location that is then linked to in the report. As is to be expected, most information is directly provided within the bug report itself, but for some features the pattern is different. *Application Code* and *Test cases* are more likely to be attached to a bug report than provided in the text. This is due to the fact that these are normally patches provided by the developers. Any automated system wishing to use these features would have to be capable of handling attachments as well as the text of the bug report.

Unsurprisingly, since only text is allowed, *Screenshots* were never provided in the bug report itself. More surprising though is the proportion that were actually provided through an external site, such as a specialised image hosting site. The reasons for this are not clear, given Bugzilla's ability to

	Obs	Exp	Rep	Bui	Cod	Err	App	Scr	Sta	Tes
Obs	-	**-0.2**	**0.3**	**0.13**	**0.16**	**0.17**	**-0.1**	**0.1**	**0.09**	0.01
Exp	-	-	**0.09**	**0.08**	-0.04	**-0.23**	**0.16**	-0.04	**-0.14**	0.03
Rep	-	-	-	**0.37**	**0.38**	0.05	**-0.12**	0.03	0.05	0.06
Bui	-	-	-	-	-0.04	0	**-0.07**	0.05	**0.13**	0
Cod	-	-	-	-	-	**0.08**	**-0.07**	-0.04	0.03	**0.11**
Err	-	-	-	-	-	-	-0.01	0.05	**0.41**	0.04
App	-	-	-	-	-	-	-	-0.04	0.04	**0.21**
Scr	-	-	-	-	-	-	-	-	-0.03	-0.01
Sta	-	-	-	-	-	-	-	-	-	0.02
Tes	-	-	-	-	-	-	-	-	-	-

Table 1: Correlations between items (significant correlations in bold)

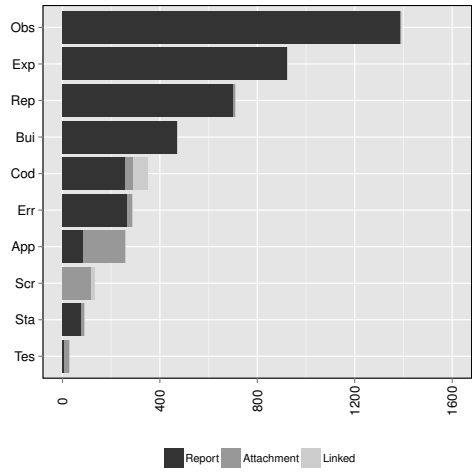

Figure 6: Number of bug reports where feature is provided, by location of provision

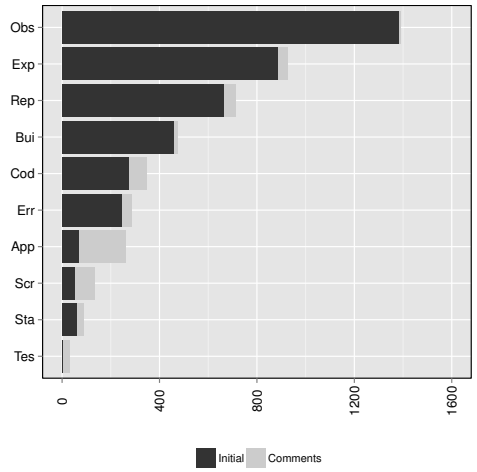

Figure 7: Number of bug reports where feature is provided, by time of provision

provide the same functionality. This was also true of *Code examples*, where in particular Firefox or Facebook bug reports would contain links to external websites or Facebook applications. This has implications for both developers and researchers wishing to make use of the information. Firstly, the sample code linked to was often more complex, consisting of an entire working site which may contain many irrelevant factors for developers, rather than a minimal working example that would be sufficient for them to reproduce the bug. In addition, the content found at external sites is not fixed, and may have changed since the bug was reported, or may in fact no longer exist at all. In Facebook Bug 3427 for example, even the *Steps to reproduce* was provided as an external link, but this link was now no longer active.

There are also a reasonable number of bugs which provide either *Code examples*, *Application Code*, *Error reports* or *Stack traces* within the main body of the report. This has important implications when parsing the contents of bug reports automatically. Although written in natural language, the bug description and comments cannot simply be treated as unstructured text. Instead, it will often contain a mixture of structured and unstructured content, and any system which wishes to handle these features will have to be able to extract them from the description body – they cannot rely on the information being provided as an attachment.

6. WHEN DO USERS PROVIDE INFORMATION?

Information from users can either be provided when the bug is initially reported, or in the comments section of the bug at a later time. For a developer, it would of course be most useful if all the information was included when the bug is first reported. This is because having to ask for comments, and waiting on responses, has a large effect on how long it takes for the bug to be fixed. Figure 7 shows when the various information was provided: when the bug is first reported, or at a later time.

Including the information when a bug is first reported is by far the most common occurrence. Overall, only 12% of the total amount of information is first provided in comments. This could be considered an encouraging result, suggesting that the original information provided in bugs is mostly sufficient. However, what is not measured is whether the information provided for each feature is *complete*. For example, a user may have provided some *Steps to reproduce*, but these may by incomplete or unclear, and a developer is still required to make additional follow-up queries. As in previous sections, no attempt has been made to verify whether the information provided in the description is complete or accurate.

There are some fields, however, where it is more com-

mon for information to not be provided until the comments, rather than in the initial description. In particular, *Application Code* and *Test cases* are far more likely to have been added later. This is largely due to the fact that these are usually added by developers working to fix the issue, rather than by the original reporter. Often in fact the code provided may be the exact patch that was applied to fix or test the issue. Any automated system which wished to assist developers in fixing bugs could therefore not rely on this being available, since by this point the developers have largely solved the issue, and the system would be redundant.

Screenshots are often also provided in a later comment, but not for this reason. Often these are provided as an attachment to the very first comment of the bug report, made by the reporter. The reasons for this are unclear, as Bugzilla allows the user to attach a file while describing the issues. This is also true to a lesser extent of other attachments such as *Error reports* or *Code examples*. In consequence, any automated system which examined only the description may well be missing valuable information.

For almost all features, there are some bugs in which the information is not provided initially. For some, such as *Stack traces* and *Error reports*, this may be because users have to be specifically prompted for such information, and are not aware of either how to retrieve it or of its importance. Unfortunately, the investigation has not captured the number of times developers ask for such information but do not receive it. This is particularly interesting for the fields mentioned because these two features are often highly accurate indicators of where in the project source code a bug is likely to be.

7. HOW DOES THE INFORMATION PROVIDED AFFECT THE OUTCOME OF THE BUG?

Sadly, not all bugs end up getting fixed by the developers – bugs can also end up being closed as invalid, duplicate or for a number of other reasons. Furthermore, the sample in this paper also included bugs that were currently being worked upon. There are two fields in Bugzilla [7] which detail the overall outcome of the bug, resolution and status. In the default Bugzilla life cycle these can take on a number of values which can in turn be customised for each project, and so in the sampled bugs a larger variety of values were present. To simplify things, all the observed values were mapped to the following smaller number of possibles outcomes:

- *Fixed*
- *Duplicate*
- *Incomplete*
- *Invalid*
- *In Progress*
- *New*

For information, the number of bugs for each bug outcome are shown in Table 2. It is interesting to note here the differences between some of the projects. Facebook in particular has a high number of *Incomplete* and *Invalid* bugs, which may reflect the behaviour stated earlier: non-technical users of Facebook using the BTS to report problems with the website, not the API. The high number of *Invalid* bugs for Apache is more surprising however, and the reasons are not clear.

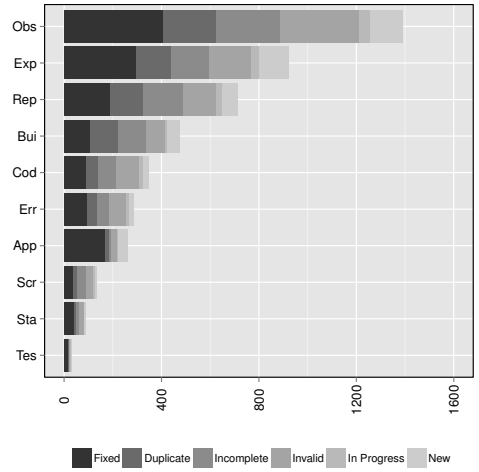

Figure 8: Number of bug reports where feature is provided, by outcome of bug

Figure 8 shows for each feature the distribution of eventual outcomes of the bugs. At first glance, there does not appear to be any strong relationship between individual features and whether or not the bug is eventually fixed. This can be confirmed by examining the correlations between the presence of each feature and the bug outcome, the significant values of which ($p<0.01$) are shown in Table 3. The only exception to this of any note is the *Application Code*, which has a correlation coefficient of around 0.33 with the outcome of *Fixed*, but this is still not especially strong. In addition since, as discussed in Section 6, *Application Code* is more likely to be provided later on, and often by the developer fixing the bug, this is hardly a useful result. It is also very surprising to note that there is in fact a very weak *negative* correlation between providing *Steps to reproduce*, *Build information*, or *Observed behaviour* and the bug being fixed.

There also does not appear to be any evidence for a relationship between the overall amount of information provided and the outcome of the bug. This can be seen by examining Fig. 9 – the range of values seen for *Fixed* bugs does not appear noticeably different to that of other bugs. It does appear that bugs which provide fewer features are more likely to be marked as *Invalid*, but again this result should not be surprising.

8. HOW COULD THIS INFORMATION BE EXTRACTED?

The sample of bugs used in this paper is of a reasonable size, but a much larger number of bugs from a wider range of projects would have to be analysed in order to even approach results which can be generalised. Given the time-consuming nature of the work, this would be tedious to perform manually, and it had been initially hoped during the study that much of the work could be automated.

For example, *Screenshots* could be identified by examin-

	Fixed	Duplicate	Incomplete	Invalid	In Progress	New
Apache	121	51	23	143	13	49
Eclipse	228	39	21	59	12	41
Facebook	89	43	122	100	23	23
Firefox	56	110	115	64	3	52

Table 2: Bug outcomes by project

	Fixed	Duplicate	Incomplete	Invalid	In Progress	New
Obs	**-0.09**	0.03	**0.08**	0.02	0.05	-0.05
Exp	0.03	0.02	-0.03	**-0.12**	0.03	**0.11**
Rep	**-0.08**	**0.09**	**0.13**	**-0.07**	-0.01	-0.03
Bui	**-0.12**	**0.18**	**0.11**	**-0.1**	-0.06	0.01
Cod	-0.06	-0.02	0.06	0.05	0.04	-0.06
Err	0.02	-0.01	0	0.01	0.04	-0.05
App	**0.33**	**-0.13**	**-0.16**	**-0.15**	0	0.06
Scr	-0.02	-0.01	**0.07**	0	0.02	-0.04
Sta	**0.07**	-0.01	-0.01	-0.02	0	-0.04
Tes	**0.09**	-0.03	-0.05	-0.04	0.05	0

Table 3: Correlation between features and bug outcomes

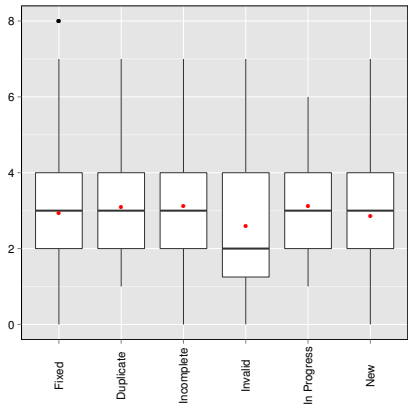

Figure 9: Range of number of features for each bug outcome

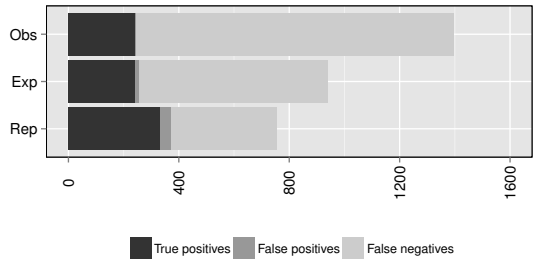

Figure 10: Automatically extracted features

ing the type of attachments or linked files. This would not be entirely accurate however; Firefox Bug 372163 contains a screenshot of how the application looks after the user's proposed patch has been applied, so is essentially a mockup. For Firefox Bug 321832, screenshots were provided, but they were embedded within a word-processing document. This may not present much of an issue for a developer, but would vastly overcomplicate automated image processing.

Previous techniques have also been developed to identify *Stack traces* [5] but these are not entirely accurate. Similar approaches may be possible for other structured features, such as *Code examples* or *Application Code*, but these are likely to be very challenging. As was shown earlier, these structured pieces of text are usually mixed liberally in with unstructured information.

Three of the most important features however – *Observed behaviour*, *Expected behaviour* and *Steps to reproduce* – are even more difficult to obtain automatically. These are unstructured features, and may or may not be explicitly marked by the user providing information. One proposal is to search for instances where the user does explicitly say they are providing a particular feature. To test this, the bug descriptions were processed by a Python script which searched for likely alternative terms or phrases, or possible abbreviations for *Observed behaviour*, *Expected behaviour* and *Steps to reproduce*. This script also ignored differences in case, spacing and minor spelling variations.

Figure 10 shows the results of attempting to automatically identify which bugs contain which of these three features, compared to the manual results. As the large number of false negatives indicates, very few of the features could be correctly identified automatically. While these may not be the most sophisticated of expressions, and more complex approaches may give better results, these do give an indication of how few bugs actually contain a regular structure indicating the features desired by developers. Investigating the positive results further revealed that the vast majority of the features that actually can be automatically identified come from Firefox, probably due to its simplified interface that specifically prompts users for these features. The unfortunate side effect is that Firefox is also responsible for most of the false positives, although there are relatively few of these. For example, in Firefox Bug 472170, what's stated as *Expected behaviour* is actually *Observed behaviour*.

9. HOW COULD THIS INFORMATION BE USED BY AUTOMATED TOOLS?

One of the key motivations behind this study was to try and address the problem of bug localisation [27, 12]. Developers are known to spend a high proportion of their time on bug fixing and maintenance - often reported to be in excess of 50% - which represents a significant cost to the software industry [21, 24]. Even before they can start working on a fix, developers often need to spend a substantial amount of time just determining where in the code the problem actually lies [8]. Bug localisation techniques aim to support developers by providing them with clues as to the whereabouts in a system of the source of a bug. For example, previous work by the authors has explored how the similarity between bug reports can be used towards this end [11], and more recently has explored how a wider range of information (such as the various sources identified in this paper) can be combined to provide the developer with a ranked list of methods, ordered by the probability that they are the source of a particular bug [10]. A key requirement of such approaches is that not only the extent to which such information is available, but, as this section explores, also the degree to which it can be automatically extracted by tools.

Steps to reproduce, Expected behaviour, Observed behaviour.

These features are all provided relatively often, and usually within the main bug description. However, the majority of the time they cannot be automatically separated from one another, or from the remainder of the bug description. As such, the most appropriate way to use these features is likely to mean keeping the description as a whole, and using information retrieval (IR) techniques to relate the bug to the source code or to find similar bugs.

Stack traces, Error reports.

These features may be simpler to extract from bug reports than many other features, although they are not entirely straightforward. However, they are not present in a large number of bugs. While they could therefore be an appropriate source of additional information to use in automated tools, they are not applicable for many bugs, and may be of most use only when used in conjunction with other sources of information.

Screenshots.

A combination of the difficulty of extracting information from *Screenshots* along with their relative scarcity mean it would be quite challenging to produce any significant bug fixing tools utilising *Screenshots*, with little applicability.

Code examples.

Code examples are not common. In addition, the language of the code is often not the same as the language of the application itself. For example, Eclipse is written in Java, and while the majority of its usage is also for Java programs, plug-ins exist for a myriad of different languages, and sample code could be in any of them. Similar issues exist for the other applications investigated in this study. Like *Screenshots* then, it may be difficult to take advantage of information from *Code examples* for automated bug fixing, and it is likely to be applicable only to a few bugs, since very few bugs provide the appropriate information.

Application Code.

Application Code was not particularly common in the given bug reports. Additionally, as the provision of such code usually indicates that the reporter has already performed some investigation of the bug on their own, utilising this information for bug fixing may not provide any significant benefit to the developers.

Test cases.

Much research has already been done on using failing test cases to determine bug locations. This work shows however that it is very rare for such tests to be provided by the user. Furthermore, many of the tests will not be provided in the same form as any existing test suite. Developers will therefore usually still be required to construct new test cases themselves in order to utilise such techniques.

Version, Build information.

The limited amount of information in these fields mean they are unlikely to provide much assistance with bug fixing on their own, but they could be used as part of a larger system.

10. THREATS TO VALIDITY

The following threats to validity were identified in relation to this work.

The *choice of projects* is limited and restricted to open (and therefore public) BTSs. Although we have tried to select a diverse set of systems, this small set cannot be regarded as representative. Furthermore, it is possible that different behaviour might be observed in a closed BTS.

The *types of information* submitted may be related to the *type of user* submitting the bug. For example, a stack trace or a test case may be more likely to be submitted by a developer than a less technically aware user and may have skewed the results.

The *analysis of information provided* is performed manually and by necessity is limited in depth. For example, if the report contained "Steps to Reproduce" then these were considered as present but the steps themselves were not actually validated. This introduces an element of subjectivity and may also influence the findings regarding the relationship between the presence of features and the outcome (for example, just because a piece of information is present does not mean it is particularly relevant or useful).

11. CONCLUSIONS

This paper has examined 1600 bugs across 4 projects and detailed how often users provide each part of a bug report. Admittedly this is from a small sample of projects, and replications and extensions of this work are necessary[2], but these are substantial and mature projects with an active user base and experienced developer teams.

From this information, it is clear that in many cases, bug reports are not complete or accurate, and often do not provide all the information that developers would wish to find,

[2]All the raw data and scripts associated with this study are obtainable from `https://personal.cis.strath.ac.uk/s.davies/`

or could use, to fix bugs. The quality of bug reports is important for developers as it impacts upon the time they take to resolve and issue, but it also has important implications for research investigating other points of the bug life cycle.

The findings from this paper also have significant implications for techniques which aim to provide some form of automated support for the problem of bug localisation. Some of the most successful techniques rely on the presence of test cases or stack traces, and these are two of the least frequently occurring items. Even if they are present in a bug report there is the additional challenge of automatically extract this information – it is also clear that most bug tracking systems were not designed with automation in mind and preponderance of free text, and variety of mechanisms for providing additional information provide yet further obstacles for any form of automated support. Finally, even though they could not be argued to be representative of all projects, another factor that any automated tool would need to take into account is how both the type of information provided and manner in which it is recorded is likely to vary between systems.

12. REFERENCES

[1] Apache HTTP. `http://httpd.apache.org`.

[2] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the International Conference on Software Engineering*, 2009.

[3] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann. What makes a good bug report? *Foundations of Software Engineering*, 2008.

[4] N. Bettenburg, R. Premraj, and T. Zimmermann. Duplicate bug reports considered harmful ... really? In *Proceedings of the International Conference on Software Maintenance*, 2008.

[5] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proceedings of the Working Conference on Mining Software Repositories*, 2008.

[6] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the Conference on Computer Supported Cooperative Work*, 2010.

[7] Bugzilla. `http://www.bugzilla.org`.

[8] S. Bugde, N. Nagappan, S. K. Rajamani, and G. Ramalingam. Global Software Servicing: Observational Experiences at Microsoft. In *Proceedings of the International Conference on Global Software Engineering*, August 2008.

[9] D. Čubranić and G. C. Murphy. Automatic bug triage using text categorization. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, 2004.

[10] S. Davies and M. Roper. Bug localisation through diverse sources of information. In *ISSRE (Supplemental Proceedings)*, pages 126–131, 2013.

[11] S. Davies, M. Roper, and M. Wood. Using bug report similarity to enhance bug localisation. In *Proceedings of the Working Conference on Reverse Engineering*, 2012.

[12] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software Maintenance and Evolution*, November 2011.

[13] Eclipse. `http://www.eclipse.org`.

[14] Facebook. `https://developers.facebook.com`.

[15] Mozilla Firefox. `http://www.mozilla.org/firefox`.

[16] GDB. `http://www.gnu.org/software/gdb/`.

[17] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proceedings of the International Conference on Software Engineering*, 2010.

[18] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the International Conference on Automated Software Engineering*, 2007.

[19] A. J. Ko and P. K. Chilana. How power users help and hinder open bug reporting. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2010.

[20] A. G. Koru and J. Tian. Defect handling in medium and large open source projects. *IEEE Software*, 21(4), 2004.

[21] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models : A Study of Developer Work Habits. In *Proceedings of the International Conference on Software Engineering*, 2006.

[22] Mozilla. `http://www.mozilla.org`.

[23] A. Schröter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Proceedings of the Working Conference on Mining Software Repositories*, May 2010.

[24] J. Singer, T. C. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the Centre for Advanced Studies Conference*, November 1997.

[25] strace. `http://sourceforge.net/projects/strace`.

[26] truss. `http://docs.oracle.com/cd/E19082-01/819-2239/truss-1/index.html`.

[27] W. E. Wong and V. Debroy. A Survey of Software Fault Localization. Technical Report UTDCS-45-09, 2009.

[28] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog : Error Diagnosis by Connecting Clues from Run-time Logs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.